Processus & signaux

Gabriel Hondet
gabriel.hondet@lsv.fr

Exercice 1 : 1.

Dans cet exercice on étudie l'utilisation et l'évaluation du code de sortie d'un processus. Normalement, le code de sortie est utilisé pour indiquer par exemple, la résussite ou non d'un programme. Ici, on s'en sert pour communiquer les résultats d'un calcul. (Remarque : le code de sortie doit entre entre 0 et 255, donc ce n'est pas idéal pour ce genre de chose).

Utilisez make pour compiler.

(a) On commence avec un exemple simple `simple.c` qui calcule la somme de deux entiers. Complétez le programme tel que le fils utilise exit pour communiquer la somme au père et que le père reçoit cette valeur par `wait`.

Solution :

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(void)
{
        int x,y;

        printf ("Donnez un entier : ");
        scanf("%d",&x);
        printf ("Donnez un autre : ");
        scanf("%d",&y);

        if (fork()) {
                pid_t s = 0;
                // obtenir la somme du fils ... à compléter
                wait(&s);
                printf("Le resultat de l'addition est %d\n", WEXITSTATUS(s));

                exit(0); // terminer
        } else {
                // Child
                int z = x+y;
                exit(z);
                // communiquer le résultat au père par code de sortie
        }
}
```

(b) Application plus complexe : télécharger l'archive `www.lsv.fr/~hondet/resources/archos/calc.zip` qui contient un calculateur simple qui évalue des expressions avec des entiers naturels, additions, multiplications et soustractions. Dans l'état actuel le programme convertit l'expression vers un arbre, puis il traverse les noeuds de l'arbre un par un pour calculer les valeurs de toutes les sous-expressions. Votre tâche est de permettre au programme de calculer les sous-expressions en parallèle. Lorsque le programme évalue une sous-expression, il devrait lancer deux processus fils qui évaluent leurs sous-expressions et qui communiquent le résultat au père par leurs codes de sortie. Le père attend les deux résultats et applique l'opération correspondante pour obtenir son propre résultat. Il (devrait) suffire de modifier la fonction `compute` dans `main`.

Solution :

```c
// main.c for calc
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <glob.h>

#include "global.h"

// name of the program, to be printed in several places
#define NAME "calc"

int compute (struct expr *expr)
{
  struct expr* to_compute = NULL;
  switch (expr->type)
  {
    case C_CONST:
      return expr->value;

    case C_PLUS:
    case C_MINUS:
    case C_MULT:
      to_compute = expr->left;
      pid_t left;
      if ((left = fork())) {
        // Lauch second process
        pid_t right;
        to_compute = expr->right;
        if ((right = fork())) {
          int status_left;
          int status_right;
          waitpid(left, &status_left, 0);
          waitpid(right, &status_right, 0);
          switch (expr->type) {
            case C_PLUS:
```

```
              return
                WEXITSTATUS(status_left) +
                WEXITSTATUS(status_right);
            case C_MINUS:
              return
                WEXITSTATUS(status_left) -
                WEXITSTATUS(status_right);
            case C_MULT:
              return
                WEXITSTATUS(status_left) *
                WEXITSTATUS(status_right);
            default:
              perror("Weird behaviour");
          }
        } else {
          exit(compute(to_compute));
        }
      } else {
        int res = compute(to_compute);
        exit(res);
      }

    default: // can't happen
      printf("something strange happened\n");
      break;
  }

  // can't happen, just to satisfy the compiler
  return 0;
}

int main (int argc, char **argv)
{
  char *prompt = malloc(strlen(NAME)+3);
  sprintf(prompt,"%s> ",NAME);

  while (1)
  {
    char *line = readline(prompt);
    if (!line) break;  // user pressed Ctrl+D; quit shell
    if (!*line) continue;  // empty line

    add_history(line);  // add line to history

    struct expr *expr = parser(line);
    if (!expr) continue;  // some parse error occurred; ignore

    printf("The result is %d!\n",compute(expr));
  }

  printf("goodbye!\n");
```

```
  return 0;
}
```

Exercice 2 : 2.

Écrire un programme qui :

- crée un processus fils ;
- transmet une séquence de bits entrés au clavier (l'utilisateur entre des 0 et des 1), bit par bit à ce processus fils.

Le processus fils doit afficher la séquence dans le bon ordre. Pour cela, utilisez des signaux. Prenez bien soin de tester plusieurs motifs de séquences (alternances de 0/1, successions de 0, successions de 1).

Pour entrer les bits, vous pouvez utiliser

```
for (c = 0; c != 0 && c != 1 && c != EOF; c = getchar());
if (c == 0) // Do something
if (c == 1) // Do something
if (c == EOF) break;
```

(a) Implémentez dans un premier temps avec l'API ANSI C (fonctions `kill`, `pause`, `signal`).

Solution :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

/* Pingpong : le père communique les '0' et '1' dans la console au fils
        en lui envoyant des signaux. On souhaite que le fils affiche tous
        les '0' et '1' rencontrés. Ctrl+D termine le programme.                */

/* Une première solution, très simple: Le parent envoie des signaux au fils
   qui attend dans une boucle while. Le grand inconvenient de cette solution
   est la consommation du processeur (à 100%, en raison de l'attente actif
   du fils). */

/***** le père *****/
int ready = 0;
void get_ready () { ready = 1; }

void parent (int cpid)
{
        char c;

        while (1)
        {
                for (c = 0; c != '0' && c != '1' && c != EOF; c = getchar());
                if (c == '0') kill(cpid,SIGUSR1);
                if (c == '1') kill(cpid,SIGUSR2);
                if (c == EOF) break;
        }
```

```
        // terminer le fils, sinon il reste vivant
        kill(cpid,SIGTERM);
        exit(0);
}

/***** le fils *****/
int recvd = -1;                 // indique la valeur envoyée par le père
void get_zero () { recvd = 0; }
void get_one () { recvd = 1; }

void child ()
{
        int ppid = getppid();

        signal(SIGUSR1,get_zero);
        signal(SIGUSR2,get_one);

        printf("parent pid = %d, child pid = %d\n",ppid,getpid());
        while(1)
        {
                /* attente active (consommation de CPU énorme) */
                while (recvd < 0);

                /* on affiche l'information reçue */
                if (recvd == 0) printf("got a 0\n");
                if (recvd == 1) printf("got a 1\n");
                recvd = -1;
        }
}

/***** programme principal *****/
int main ()
{
        int cpid = fork();
        if (cpid) parent(cpid);
        else child();
        return 0;
}
```

(b) puis avec l'API Posix.

Solution :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

/* Le programme suivant résout le problème de façon définitive :
   on remplace pause par sigsuspend. Ce dernier permet de mieux
```

```
   contrôler la livraison des signaux. Ici, ceux-ci ne peuvent
   intervenir que pendant un appel de sigsuspend ce qui élimine
   le problème précédent. */

static sigset_t newmask, zeromask;

/***** le père *****/

int ready = 0;
void get_ready () { ready = 1; }

void parent (int cpid)
{
        char c;

        /* Installer le gestionnaire pour que le fils
           puisse signaler qu'il est prêt. */
        struct sigaction action;
        memset(&action,0,sizeof(action));
        action.sa_handler = get_ready;
        sigaction(SIGUSR1,&action,NULL);

        while (1)
        {
                /* On attend le signal du fils avant d'envoyer le prochain
                   signal. Notons que sigsuspend sera débloqué par n'importe
                   quel signal, alors on utilise ready pour assurer que c'est
                   bien le signal USR1 qui a été déclenché. */
                while (!ready) sigsuspend(&zeromask);
                ready = 0;

                /* On ignore les charactères autre que 0, 1 et EOF. */
                for (c = 0; c != '0' && c != '1' && c != EOF; c = getchar());
                if (c == '0') kill(cpid,SIGUSR1);
                if (c == '1') kill(cpid,SIGUSR2);
                if (c == EOF) break;
        }
        kill(cpid,SIGTERM);        /* terminer le fils */
        exit(0);
}

/***** le fils *****/

int recvd = -1;
void get_zero () { recvd = 0; }
void get_one () { recvd = 1; }

void child ()
{
        int ppid = getppid();
```

```
        /* installer les gestionnaires des deux signaux */
        struct sigaction action;
        memset(&action,0,sizeof(action));
        action.sa_handler = get_zero;
        sigaction(SIGUSR1,&action,NULL);
        action.sa_handler = get_one;
        sigaction(SIGUSR2,&action,NULL);

        printf("parent pid = %d, child pid = %d\n",ppid,getpid());
        while(1)
        {
                /* signaler au père qu'on est prêt */
                kill(ppid,SIGUSR1);

                /* le même principe que pour le père */
                while (recvd < 0) sigsuspend(&zeromask);
                if (recvd == 0) printf("got a 0\n");
                if (recvd == 1) printf("got a 1\n");
                recvd = -1;
        }
}

/***** programme principal *****/
int main ()
{
        /* Tout d'abord, avant le fork(), on va bloquer les deux signaux.
           Ce blocage sera hérité par le père et le fils, ce qui assure
           que aucun signal n'est raté quand les processus démarrent. */
        sigemptyset(&zeromask);
        sigemptyset(&newmask);
        sigaddset(&newmask,SIGUSR1);
        sigaddset(&newmask,SIGUSR2);
        sigprocmask(SIG_BLOCK, &newmask, NULL);

        /* Ensuite, on appelle fork. */
        int cpid = fork();
        if (cpid) parent(cpid);
        else child();
        return 0;
}
```

Exercice 3 : 3.

Programmation système et λ calcul ne sont pas incompatibles : le dialecte Guile du Scheme implémente les API Posix. Écrivez un programme (en Guile) qui

- se fork
- le père attend un certain temps puis tue son fils (`SIGINT`),
- le fils imprime des citations sur la sortie standard avec la commande `fortune -e` (`-o` pour être insultant) et une pause entre chaque sortie.

Vous aurez besoin de la doc Guile sur Posix `https://www.gnu.org/software/guile/manual/html_node/POSIX.html`.

Comme point de départ :

```
#!/usr/bin/guile \
-e main -s
!#
(define (main args)
  (let ((pid (primitive-fork)))
    (if (= pid 0)
        (...)
        (...))))
...
```

le fichier peut être rendu exécutable puis être interprété.

Vous pouvez bien entendu l'implémenter d'abord en C, pour apprécier le `execl` ou `execlp`.

Solution :

```
#!/usr/bin/guile \
-e main -s
!#

(define (main args)
  (let ((pid (primitive-fork)))
    (if (= pid 0)
        (child)
        (begin
          (sleep 3)
          (kill pid SIGINT)))))

(define (child)
  (let ((pid (primitive-fork)))
    (if (= pid 0)
        (write-bullshit)
        (begin
          (sleep 1)
          (child)))))

(define (write-bullshit)
  (execl "/usr/games/fortune" "-e"))
```